Module 2: The Main Engine of ModelingToolkit.jl

Madhusudhan Pandey, PhD

November 20, 2025

Abstract

This lecture provides an overview of the standard repository structure used in modern scientific Julia packages, using `ModelingToolkit.jl` as a representative example. The session introduces graduate and PhD students to the essential folders, configuration files, and automation tools that support high-quality research software development. Key concepts include documentation workflows, benchmarking, modular extensions, testing infrastructures, and continuous integration pipelines. By understanding the purpose and interplay of components such as `src`, `docs`, `test`, and the `.github` automation ecosystem, students gain the foundational skills required to design, maintain, and publish professional scientific software. The lecture equips learners to build reproducible, scalable, and maintainable research codebases aligned with best practices in computational science and engineering.

This document is part of a multi-module series. This is **Lecture Module 2**. Each module in this lecture series is designed as a self-contained 30-minute session. The material may be taken independently or with an instructor, and is suitable for learners at the beginner, intermediate, or advanced level. This flexibility allows students to progress at their own pace while ensuring that each module provides a complete and accessible learning experience.

1 Introduction

ModelingToolkit.jl (MTK) is a large, multi-component scientific package whose internal organisation illustrates the architectural patterns typical of high-quality Julia research software. Understanding its internal structure helps students learn how real scientific packages are engineered.

In Module 1, we surveyed the repository layout, including key folders such as `src/`, `docs/`, `benchmark/`, `test/`, and automation scripts in the `.github` directory. In this Module 2, we focus on the **core file that binds the entire package together:**

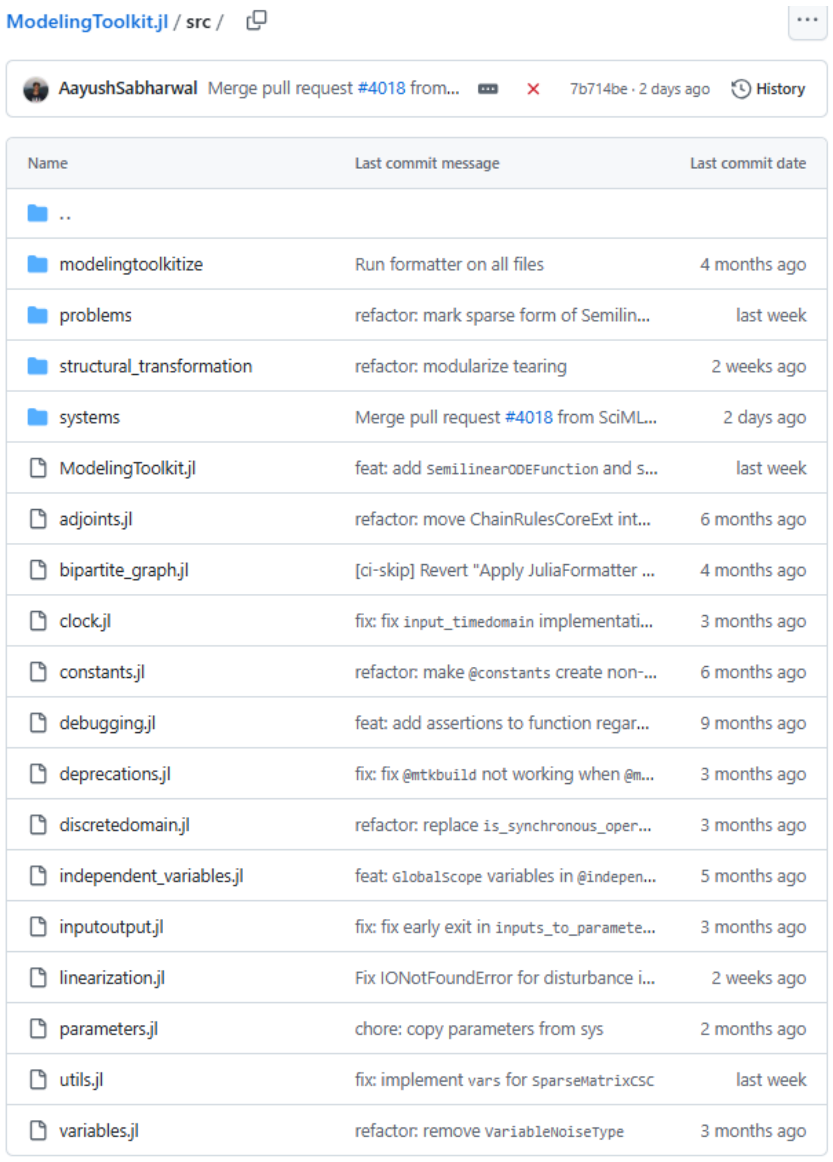
`src/ModelingToolkit.jl`

This file works like the main engine of the package. It connects all the important parts—symbolic types, system builders, equation tools, transformations, compiler steps, and the functions that the user can access. Figure 1 shows files and folders that are in the same path as the main file *ModelingToolkit.jl*.

2 Why `src/ModelingToolkit.jl` Matters

Although MTK contains dozens of source files, the `src/ModelingToolkit.jl` file serves as the unifying layer. It performs four essential functions:

1. **Declares the module** and establishes the MTK namespace.
2. **Imports dependencies** required by symbolic types, structural transformations, solvers, and compiler routines.


ModelingToolkit.jl / src /

AayushSabharwal Merge pull request #4018 from... 7b714be · 2 days ago History

Name	Last commit message	Last commit date
..		
modelingtoolkitize	Run formatter on all files	4 months ago
problems	refactor: mark sparse form of Semilin...	last week
structural_transformation	refactor: modularize tearing	2 weeks ago
systems	Merge pull request #4018 from SciML...	2 days ago
ModelingToolkit.jl	feat: add `SemilinearODEFunction` and s...	last week
adjoints.jl	refactor: move ChainRulesCoreExt int...	6 months ago
bipartite_graph.jl	[ci-skip] Revert "Apply JuliaFormatter ...	4 months ago
clock.jl	fix: fix `input_timedomain` implementati...	3 months ago
constants.jl	refactor: make `@constants` create non-...	6 months ago
debugging.jl	feat: add assertions to function regar...	9 months ago
deprecations.jl	fix: fix `@mtkbuild` not working when `@m...`	3 months ago
discretedomain.jl	refactor: replace `is_synchronous_oper...`	3 months ago
independent_variables.jl	feat: `GlobalScope` variables in `@indepen...`	5 months ago
inputoutput.jl	fix: fix early exit in `inputs_to_paramete...`	3 months ago
linearization.jl	Fix IONotFoundError for disturbance i...	2 weeks ago
parameters.jl	chore: copy parameters from sys	2 months ago
utils.jl	fix: implement `vars` for `SparseMatrixCSC`	last week
variables.jl	refactor: remove `VariableNoiseType`	3 months ago

Figure 1: Folders and files that are in the same path to the main file *ModelingToolkit.jl*.

3. **Exports the user-facing API**, defining what users access when they write `using ModelingToolkit`.

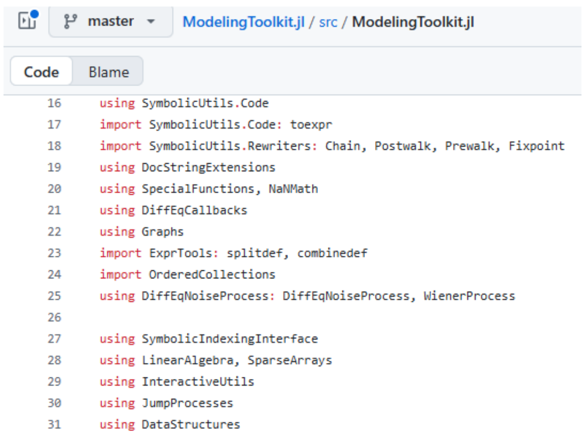
4. **Includes** all internal component files, stitching together types, systems, utilities, compiler passes, and symbolic processing tools.

This architecture ensures a coherent outer interface while allowing the internal codebase to remain modular, readable, and maintainable.

3 How a Julia Module is Generally Constructed

The full source file of `ModelingToolkit.jl` contains hundreds of imports, exports, and `include` statements. Although the file is long, the underlying idea behind how a Julia module is constructed is simple. Figure 4 shows imports and exports of the module *ModelingToolkit*. A Julia module is built around a few core principles:

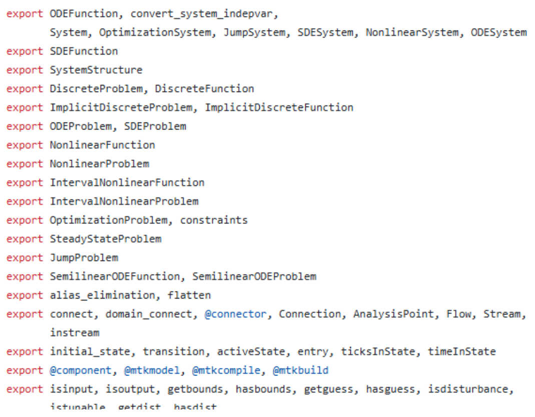
1. **Start a namespace using `module`** Every package begins with a line such as:



```julia
using SymbolicUtils.Code
import SymbolicUtils.Code: toexpr
import SymbolicUtils.Rewriters: Chain, Postwalk, Prewalk, Fixpoint
using DocStringExtensions
using SpecialFunctions, NaNMath
using DiffEqCallbacks
using Graphs
import ExprTools: splitdef, combinedef
import OrderedCollections
using DiffEqNoiseProcess: DiffEqNoiseProcess, WienerProcess

using SymbolicIndexingInterface
using LinearAlgebra, SparseArrays
using InteractiveUtils
using JumpProcesses
using DataStructures
```

Figure 2: (a) Imported packages in *ModelingToolkit module.*



```julia
export ODEFunction, convert_system_indepvar,
       System, OptimizationSystem, JumpSystem, SDESystem, NonlinearSystem, ODESystem
export SDEFunction
export SystemStructure
export DiscreteProblem, DiscreteFunction
export ImplicitDiscreteProblem, ImplicitDiscreteFunction
export ODEProblem, SDEProblem
export NonlinearFunction
export NonlinearProblem
export IntervalNonlinearFunction
export IntervalNonlinearProblem
export OptimizationProblem, constraints
export SteadyStateProblem
export JumpProblem
export SemilinearODEFunction, SemilinearODEProblem
export alias_elimination, flatten
export connect, domain_connect, @connector, Connection, AnalysisPoint, Flow, Stream,
       instream
export initial_state, transition, activeState, entry, ticksInState, timeInState
export @component, @mtkmodel, @mtkcompile, @mtkbuild
export isinput, isoutput, getbounds, hasbounds, getguess, hasguess, isdisturbance,
       istunable, getdist, hasdist
```

Figure 3: (b) Exported functions from the *ModelingToolkit* module that can be access using *using ModelingToolkit*

Figure 4: Import and export of *ModelingToolkit* module.

```julia
module ModelingToolkit
```

This creates a self-contained environment where all functions, types, and submodules live.

2. **Load dependencies using `using` and `import`** Large scientific packages require many external tools. At the top of the file, the module brings these tools into scope:

   - symbolic manipulation packages,
   - linear algebra tools,
   - graph utilities,
   - differential equation interfaces,
   - code-generation helpers,
   - compiler and transformation libraries.

   In the actual `ModelingToolkit.jl` file, this list is long, but conceptually it simply means: *"Bring in everything this module needs."*

3. **Declare the package's public API using `export`** The module then lists all functions, macros, and types that should be visible to the user when they write:

   ```julia
   using ModelingToolkit
   ```

   This export list defines the user-facing interface of the package.

4. **Include all internal source files using `include`** Instead of writing the entire codebase in a single file, Julia packages split the implementation into many files, such as:

   ```julia
   include("variables.jl")
   include("systems/odesystem.jl")
   include("problems/odeproblem.jl")
   include("structural_transformation/...")
   ```

The main file stitches these components together:

> *"Load all the pieces that implement variables, systems, solvers, code generation, and transformations."*

5. **Define types, functions, and macros inside the module** A module contains:
   - abstract types (e.g., `AbstractSystem`),
   - concrete system types (e.g., `ODESystem`, `SDESystem`),
   - helper functions,
   - macros such as `@variables`, `@parameters`, `@component`,
   - symbolic transformations and compiler routines.

   These definitions collectively form the "intelligence" of the package.

6. **End the module with `end`** Every module closes with:

   ```
   end # module
   ```

   marking the end of its namespace.

The fundamental structure of a Julia module is simple:

> **Start a module, import what you need, define the public API, include the internal files, implement the logic, and then close the module.**

This pattern is the same whether the module is small (a few files) or extremely large like ModelingToolkit.jl.

4 Core Responsibilities of `ModelingToolkit.jl`

The main engine file performs several high-level responsibilities that drive the functionality of the entire package:

4.1 Namespace and Module Declaration

The file begins with:

```
module ModelingToolkit
```

This creates the namespace within which all MTK functions, types, macros, and submodules exist.

4.2 Importing Fundamental Dependencies

MTK requires packages from the SciML ecosystem, Julia Base, Symbolics, graph utilities, structure analysis tools, and differential equation preprocessors. These imports support symbolic expression manipulation, DAE indexing, linearisation, code lowering, and system construction.

4.3 Exporting the Public API

A critical function of the main engine is to declare the public interface of MTK. The export block lists types such as `ODESystem`, `DAESystem`, `@variables`, `@parameters`, and dozens of transformation utilities. This is what users see when they load ModelingToolkit.

4.4 Including Component Files

The engine file contains many `include(...)` statements that pull in the sub-files implementing:

- variable and parameter constructors,
- system definitions (ODE, DAE, SDE, Nonlinear),
- structural transformations,
- symbolic utilities,
- simplification passes,
- equation manipulation functions,
- compilation and lowering helpers,
- IO and display logic.

Although these components live in separate files, the main engine unifies them into a single namespace.

4.5 Binding Symbolic Systems to Compiler Passes

The most important role of `ModelingToolkit.jl` is to map high-level symbolic expressions and system definitions to the compiler pipeline that:

1. analyses system structure,
2. performs tree walks and graph transformations,
3. reduces DAEs,
4. constructs Jacobians,
5. performs symbolic simplification,
6. generates optimized IR (intermediate representations),
7. prepares the system for downstream solvers.

Without the coordinating engine file, this pipeline would be fragmented across dozens of source files.

5 A Conceptual View of the Engine

The main engine file performs the following conceptual tasks:

- **Integration:** Connects all symbolic, structural, and system-level components.
- **Organisation:** Defines where each submodule belongs in the package hierarchy.
- **Presentation:** Exposes a unified API even though functionality is internally distributed.
- **Stabilisation:** Ensures transformations and system constructors interact correctly.

It is the “glue” that binds the ModelingToolkit ecosystem together.

6 Summary

The file `src/ModelingToolkit.jl` is the central coordinating element of the entire package. It declares the module, exposes the API, includes component files, and defines the symbolic modeling environment. It is the engine that binds together:

- symbolic system constructors,
- differential-algebraic formulations,
- compiler passes,
- structural transformations,
- linearisation mechanisms,
- simplification utilities.

Understanding this file is essential for students studying how large-scale Julia packages are architected and how symbolic modeling tools are engineered.